$\overline{\textbf{Off}}\underline{\textbf{line}}$ Software User's Guide

S. Argirò[a]
S.L.C. Barroso and L. Prado Jr[b]
T. McCauley and T. Paul[c]
L. Nellen[d]
T. Porter[e]
M. Roth[f]
D. Veberič[gh]

[a] University of Torino and INFN
[b] Universidade Estadual de Campinas
[c] Northeastern University
[d] Universidad Nacional Autonoma de Mexico
[e] Louisiana State University
[f] IEKP, Universität Karlsruhe
[g] Nova Gorica Polytechnic, Slovenia
[h] LAL, IN2P3/CNRS & Uni. Paris-Sud

GAP-2005-XXX

Contents

1 Introduction **5**

2 Documentation and Resources **6**

3 Getting Started **6**

3.1 Installation 6

3.2 Developing your own project using the framework 7

3.2.1 Preparing your module 7

3.2.2 Including your module in a sequence 8

3.2.3 Configuring your project 8

4 Examples **9**

4.1 SSimulation 10

4.2 SReconstruction 13

4.3 FSimulation 14

4.4 FReconstruction 15

5 Framework Design and Use **16**

5.1 Event 16

5.1.1 Examples of event interface use 17

5.2 Detector 17

5.2.1 Detector configuration 18

5.2.2 Atmosphere description 18

5.3 Sequencing Modules and the RunController 18

5.4 Configuration files and CentralConfig 22

5.4.1 Using the CentralConfig 22

5.4.2 Configuration logging 24

5.4.3 Default configurations 24

5.5 Reading an XML file from your module 26

5.5.1 Accessing your configuration file 26

5.5.2 Navigating Data Organized in Trees 26

5.5.3 Validating configuration files 29

5.6 Databases 31

5.7 Geometry 31

5.8 Error logging 31

5.9 Time stamps and time intervals 31

A Appendix: The Offline Build System **33**

B Appendix: Installation and configuration of the GEANT4-based SD simulation **33**

1 Introduction

This document provides a brief introduction to the Offline framework. More detailed documentation can be found in the sources described in section 2. The framework provides core machinery to help in writing SD, FD and hybrid simulation and reconstruction codes. In the past, such functionality was scattered across half a dozen or so different packages, each of which took a somewhat different approach to the problems, making it difficult and time consuming to glue them all together or to compare physics algorithms. The code described here was prepared from the ground up in an attempt to meet the requirements of all the various simulation and reconstruction tasks, and to thus make it easier to share code and collaborate on the physics problems. Details of framework design is presented elsewhere [1] [2] [3].

Briefly, the Offline framework offers the following services:

- machinery for modularizing physics algorithms so that different algorithms can be easily compared and/or arranged in different ways to carry out various tasks,
- structures for building up simulated and reconstructed events and handling persistency (storing on disk), including the ability to read from and write to a variety of existing file formats (AIRES, CORSIKA, CDAS, FDEventLib).
- access to information on the detector configuration and performance as a function of time, including data stored in MySQL databases,
- machinery for managing configuration files,
- a collection of utilities to assist in geometrical manipulations, parsing files, logging error messages, and handling units, as well as a number useful classes representing traces, time intervals and time stamps, tabulated functions, particles, and so forth.
- a simple-to-use and robust build system based on the GNU autotools,

Figure 1 indicates the principal components of the framework. Simulation and reconstruction tasks are factorized into so-called modules, each of which is able to read information from the detector description and/or the event, process the information, and write information back into the event. The detector description provides access to the information on detector performance and configuration. The event facilitates communication between modules, and part or all of it can be stored to disk under direction of the user. Modules are sequenced under command of a RunController, with the sequence defined by the user in an external file.

This document is organized as follows. Section 2 indicates where you can find additional information and resources useful for installing the software and finding out more about design details and new developments. Section 3 uses a very simple example to familiarize you with the main ingredients of the Offline software; details are deferred to later sections. Section 4 contains brief descriptions of the example simulation and reconstruction applications which are distributed with the software. Finally, section 5 provides more detailed information on the parts of the Offline code introduced in section 3.

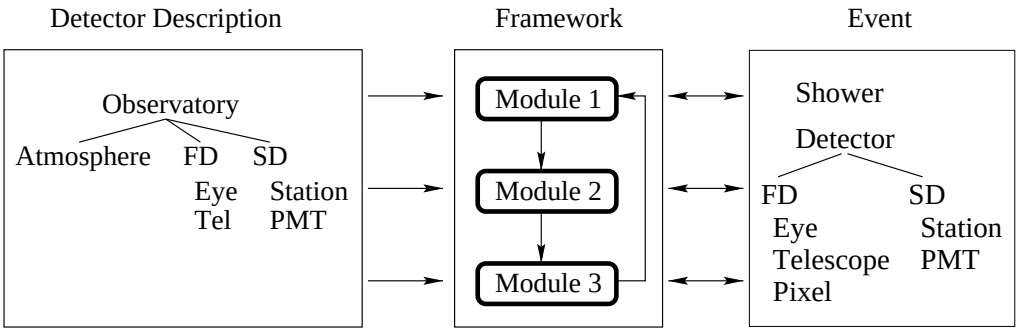


Figure 1: Simulation and reconstruction tasks are broken down into modules. Each module is able to read information from the detector description and/or the event, process the information, and write the results back into the event. Communication between modules occurs only through the event.

2 Documentation and Resources

Up–to–date documentation, discussion forums and bug tracking are available on the **Offline** web pages:

- The main portal to all web-based **Offline** information is
  `http://www.auger.unam.mx/AugerWiki/OfflineSoftware`
- Bugs and issues can be searched for and reported at
  `http://www.auger.unam.mx/bugzilla/`
- You can sign up with various framework-related mailing lists and read discussion archives at `https://www.auger.unam.mx/mailman/listinfo/`
- You can browse the Offline CVS repository and find the latest tagged versions at:
  `http://auger.in2p3.fr/cgi-bin/cvsweb.cgi/Offline/`
- Regular builds of Offline doxygen source documentation can be found at:
  `http://offline.p-ng.si/`

3 Getting Started

This section explains how to install the framework, provides a brief introduction to the basic components of a framework-based project, and describes some of the examples distributed with the framework.

3.1 Installation

The Offline code and the required external packages are available for installation either from sources or precompiled binaries. You can download everything you need from this web site:
`http://www.ifi.unicamp.br/AUGER/projects/dpa/software/distribution.html`

At this site, you will also find step–by–step instructions on carrying out the framework installation and setting up local databases. This page also provides instructions on installing directly from the CVS repository.

Normally it should only be necessary to have a single installation of the framework and a single copy of the databases per institute. It is probably best if one computer-savvy person performs the installation and then notifies collaborators at his or her institute where the framework was installed. Individual physicists can then develop code in their own areas and link it with the framework. Section 4 will point you to some examples of how to do this.

3.2 Developing your own project using the framework

Here we present a short description of the steps involved in setting up your own project in the framework. More details can be found in the examples distributed with the framework (see section 4), or at one of the web sites described in section 2. Several examples are located in:
`$prefix/share/auger-offline/doc`
where `$prefix` is the directory where the framework is installed. The discussion below approximately follows the simple example found in

`$prefix/share/auger-offline/doc/UserSkeleton`

and describes the main ingredients you will find in that example.

3.2.1 Preparing your module

Your can introduce your own algorithms into a simulation or reconstruction chain by putting them into a so-called user module. A user module is simply a C++ class that fulfills a few requirements that make it possible to plug it in to the framework. This plugging-in is facilitated through inheritance from an interface class called `fwk::VModule`, as depicted in Figure 2. The `fwk::VModule` class provides a few services and, most importantly, defines three functions that you must implement in your module. These functions are : `Init()`, `Run(evt::Event& event)`, and and `Finish()`. The first and last are called once per run of the program, while the second is called once each time a module is looped over. In the `Init()` method you might put whatever code you need to configure your module, read in data, book histograms and so forth. The `Run(evt::Event& event)` method should contain whatever processing you want to do for each event. The current `Event` is available through the reference passed as an argument of the `Run` function. Writing out histograms and cleanup can be put in the `Finish()` function.

To make the framework aware that your module exists, you need only to add

```
REGISTER_MODULE("MyModule",MyModule);
```

at the end of your class definition. This macro specifies the class name for your module (second argument), and assigns it an identifier name by which it is known to the outside world (first argument). The identifier name can be the same as the class name, but does not have to be. To give a concrete example, the class declaration of a user module might look something like figure 2.

```
class MyModule : public fwk::VModule{

public:

  MyModule();
  virtual ~MyModule();

  fwk::VModule::ResultFlag Init();
  fwk::VModule::ResultFlag Run(evt::Event & event);
  fwk::VModule::ResultFlag Finish();

private :

  REGISTER_MODULE("MyModule",MyModule);
};
```

Figure 2: Example class declaration for a Module. The `fwk::VModule::ResultFlag` return values are explained in section 5.3.

3.2.2 Including your module in a sequence

Once you've prepared a module, you need to instruct the framework on how to use it. Such instructions are written in a so-called sequence file, which is read in by the framework and used to call up the modules you wish to use in a particular order. Your module is specified in the sequence file by its identifier name. For example, a sequence file which twice calls the `Run()` method of module "MyModule" is shown in Figure 3. To add more modules into a sequence, simply write

```
<!-- Trivial sequence file that loops over MyModule Run method twice -->

<sequenceFile>
  <moduleControl>
    <loop numTimes="2">
       <module> MyModule </module>
    </loop>
  </moduleControl>
</sequenceFile>
```

Figure 3: A simple module sequence file.

them in the sequence file in the order you want them called, from top to bottom. Loops are specified with a `<loop numTimes="n">` tag at the beginning of the loop and a `<\loop>` tag at the end. More details on controlling modules sequences can be found in section 5.3.

3.2.3 Configuring your project

Finally you need to tell the framework where to find any configuration files it may need. Links to configuration files are put in a so-called bootstrap file, which is usually called `bootstrap.xml`. For the trivial example above (section 3.2.2), the bootstrap file must contain, at minimum, the path to the sequence file. You might also want to define a path to a configuration file for "My-

Module". If you had a sequence file `ModuleSequenceExample.xml` and a module configuration file `MyModuleConfig.xml` in the directory with the executable, the bootstrap file might look like the one in figure 4.

```
<bootstrap xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation='/someplace/bootstrap.xsd'
           xmlns:xlink="http://www.auger.org/schema/types">

  <centralConfig>

    <configLink
        id         = "ModuleSequence"
        type       = "XML"
        xlink:href = "./ModuleSequenceExample.xml" />

    <configLink
        id         = "MyModuleConfig"
        type       = "XML"
        xlink:href = "./MyModuleConfig.xml" />

  </centralConfig>
</bootstrap>
```

Figure 4: A bootstrap.xml file for a trivial application. The text following `<bootstrap` is explained in section 5.4.

When you run the executable, the name of the bootstrap file is passed on the command line:

```
userAugerOffline -b bootstrap.xml
```

The framework parses this bootstrap file and constructs a dictionary associating each `id` declared in the bootstrap file to a so-called `xlink:href`, which just points to a physical file on a disk somewhere [1]. In Figure 4, a logical id called "ModuleSequence" points to a file in the local directory called "ModuleSequenceExample.xml". Code belonging to the core framework or to physics modules can then request a particular configuration file by specifying its logical id, as explained in detail in section 5.4.1. This makes it easy to change configurations; you simply edit the bootstrap file and point the logical id to the configuration file you wish to use.

In Figure 4, the purpose of the mysterious text following `<bootstrap` and preceding the next `>` is explained in section 5.4. More details on program configuration, XML and related topics appear in sections 5.4 and 5.5.

4 Examples

The $\overline{\text{Off}}\underline{\text{line}}$ distribution includes several examples you can emulate when starting your own project. A minimal skeleton of what you will need to prepare your own module and link it into the framework is available in the

[1] The odd-looking xlink:href notation is used in order to adhere to the XML standard. See section 5.5 for more information about XML.

`$prefix/share/auger-offline/doc/UserSkeleton`
directory, where `$prefix` is the place where the framework is installed at your institute. This skeleton contains a trivial example module, a module sequencing file, a simple `bootstrap.xml` file, and a Makefile to compile the example Module and link it with the framework. You can use this skeleton as a template for your own framework-based projects. Simply copy it from the installation directory into your own area, and modify as needed. Section 3.2 gives a brief description of the main ingredients in a framework-based project.

There are more involved examples available in the
`$prefix/share/auger-offline/doc/Examples`
directory. These examples demonstrate realistic simulation and reconstruction chains assembled out of `Module`s distributed as a part of the framework as well as special modules included in the example directories. To try the examples, copy them to your own area, run `make` in the directory for the example you are interested in, and type `userAugerOffline -b bootstrap.xml`. The module included with the example will generally produce a `.root` file which you can view with the ROOT browser. More details on the examples are given below.

4.1 SSimulation

The `SSimulation` example generates simulated surface detector events. All modules used in this sequence are configurable via their individual `xml` configuration files. Most are provided with reasonable defaults, but the user is encouraged to vary the input parameters for modules to get a feeling for the effect they have on the results.

The module sequence for this example is :

```
<sequenceFile>

   <moduleControl>

    <loop>

       <loop numTimes="100" save="no">

       <module> ParticleInjectorOG </module>
       <module> FastTankSimulatorOG </module>
       <module> SdPMTSimulatorOG </module>
       <module> SdFilterFADCSimulatorMTU </module>
       <module> SdBaselineSimulatorOG </module>
       <module> SdSimulationCalibratorOG </module>

       </loop>

       <loop numTimes="1" save="yes">
```

```
        <module> EventFileReaderOG </module>

        <loop numTimes="1" save="yes">

           <module> EventGeneratorOG </module>
           <module> ShowerRegeneratorOG </module>
           <module> FastTankSimulatorOG </module>
           <module> SdPMTSimulatorOG </module>
           <module> SdFilterFADCSimulatorMTU </module>
           <module> SdBaselineSimulatorOG </module>
           <module> TankGPSSimulatorOG </module>
           <module> TankTriggerSimulatorOG </module>
           <module> CentralTriggerSimulatorOG </module>
           <module> UserModule </module>
           <module> EventFileExporterOG  </module>

        </loop>

      </loop>

    </loop>

   </moduleControl>

</sequenceFile>
```

In this example, we have an initial 'calibration' loop, and then a nested inner loop that performs the actual detector simulation. The calibration loop is performed by the sequence

```
<loop numTimes="100" save="no">

<module> ParticleInjectorOG </module>
<module> FastTankSimulatorOG </module>
<module> SdPMTSimulatorOG </module>
<module> SdFilterFADCSimulatorMTU </module>
<module> SdBaselineSimulatorOG </module>
<module> SdSimulationCalibratorOG </module>

</loop>
```

This module sub-sequence corresponds to injecting 1 particle into a tank, running the tank/electronic simulation sequence that will be used in the detector simulation to follow, and recording the appro-

priate calibration information using the SdSimulationCalibratorOG module. To build statistics, the loop is executed 100 times (signified by the numTimes = 100 loop tag parameter), without resetting the event to its initial state at loop entry (specified by the save="no" loop tag). It is important not to reset the event to its state at loop entry because the cumulative information will not be available at the completion of the calibration pass.

Following the calibration phase, the simulation sequence itself is executed. The module subsequence for this is

```
<loop numTimes="1" save="yes">

  <module> EventFileReaderOG </module>

  <loop numTimes="1" save="yes">

  <module> EventGeneratorOG </module>
  <module> ShowerRegeneratorOG </module>
  <module> FastTankSimulatorOG </module>
  <module> SdPMTSimulatorOG </module>
  <module> SdFilterFADCSimulatorMTU </module>
  <module> SdBaselineSimulatorOG </module>
  <module> TankGPSSimulatorOG </module>
  <module> TankTriggerSimulatorOG </module>
  <module> CentralTriggerSimulatorOG </module>
  <module> UserModule </module>
  <module> EventFileExporterOG  </module>

  </loop>

</loop>
```

In this part of the total sequence the actual detector simulation is done, and results are output to the various (user selected) file formats. At entry to the outermost loop in this sub-sequence, the state of the event after calibration is preserved by using the save = "yes" loop tag. This ensures that, apart from the previously calculated calibration information, the event is cleared. The EventFileReaderOG module is then used to read in the AIRES shower included in the example directory. Then, the innermost loop is entered, again preserving the state of the event using the save = "yes" tag, but now with the simulated shower information included. The module sequence for the detector simulation follows a logical progression : the EventGeneratorOG module places the core of the shower somewhere over the array which the ShowerRegeneratorOG module then regenerates to give lists of injected particles into the tanks. The FastTankSimulatorOG-SdPMTSimulatorOG-SdFilterFADCSimulatorMTU-SdBaselineSimulatorOG sequence of modules

performs the simulation of particles in the tanks, giving FADC traces with baselines added. The `TankGPSSimulatorOG` provides tank GPS times, and the `TankTriggerSimulatorOG` performs the local triggering for each tank hit by particles. The `CentralTriggerSimulatorOG` performs the central trigger simulation; it is important to note that the same code used in the event builder in CDAS is also used within this module. Finally, the `UserModule` and `EventFileExporterOG` modules output the simulated event into different file formats. As the example is currently configured, the `UserModule` outputs results as plain `ROOT` histograms and graphs viewable using a `TBrowser`, and the `EventFileExporterOG` module exports into CDAS data format. Assuming the user has not changed the output filenames for the `UserModule` or `EventFileExporterOG` from the example defaults, the simulation output is available in the files `OfflineOutput.root` and `Output.root` respectively.

In this example, it is straightforward to compare the results of different tank simulations. To do this, the user can replace the `FastTankSimulatorOG` with `G4TankSimulatorOG`, `SDSimTankSimulatorOG`, or other tank simulators that may become available. Similarly, different electronics modules can be interchanged as they become available. Again, it is important to emphasise that if these changes are done, the appropriate simulator modules must be used in the calibration and simulation passes of the total module sequence. Additionally, variations in the parameters of the `ShowerRegeneratorOG` module can give the user insight into the effect of the regeneration algorithm on thinned showers, and the regenerated particle lists in tanks.

4.2 SReconstruction

The `SReconstruction` example provides some basic information gathered from the SD reconstruction modules, i.e. `SdPlaneFitOG` and `LDFFinderOG`. In addition the sequence file invokes:

```
<moduleControl>
   <loop numTimes="unbounded">
      <module> EventFileReaderOG  </module>
      <module> SdCalibratorOG     </module>
      <module> SdEventSelectorOG  </module>
      <module> SdStationSelectorOG  </module>
      <module> SdPlaneFitOG       </module>
      <module> LDFFinderOG        </module>
      <module> UserModule         </module>

      <module> SdRecPlotterOG     </module>
   </loop>
</moduleControl>
```

SD data are read by the `EventFileReaderOG` and calibrated by `SdCalibratorOG`. If neccessary further analysis code can be put piggyback in the `UserModule`, when the angular and LDF fits are done. Some example statements how to access interesting information are given there. When the example is run, the file `OfflineOutput.root` will be generated in the same directory, and it

will contain some plots of the results, i.e. an LDF fit.

The angular 'fit' done in the module `SdPlaneFitOG` is a simple analytic solution of the χ^2 minimization for the planar shower-front geometry model.

The `LDFFinderOG` module calculates barycentre and perfoms the fitting of angle of incidence (shower axis), S_{1000}, LDF slope parameters (β, γ), and shower-front curvature, iteratively. The fits are done in several stages according to quality conditions given in the `LDFFinder.xml`. The definition of the two slope parameters, β and γ, depend on the acctual choice of the LDF parameterisation. The fit procedure is as follows

Stage 0 calculation of (square-root signal) weighted barycenter and bary-time, first estimation of β, γ, and S_{1000},

Stage 1 fitting of candidate stations for S_{1000} and impact point (core) position,

Stage 2 fitting of candidate and silent stations for S_{1000} and core position,

Stage 3.1 analytic estimation of the shower-front curvature and axis,

Stage 3.2 fitting of candidate stations for shower-front curvature and axis.

Selected in the XML configuration file, a likelihood or χ^2 method can be performed to estimate the LDF parameters. The default LDF parameterization is taken from [5].

4.3 FSimulation

The `FSimulation` example illustrates FD simulations in the Offline from shower particles in the atmosphere up to FADC traces in the telescopes. The starting point is the reading or simulation of the shower data, namely, the shower longitudinal charge profile and energy deposit, to be used in the next simulation steps. To start the process by reading simulated data generated either from AIRES or from CORSIKA programs, the first module to be used in the sequence is the `EventFileReaderOG`. Alternatively, the sequence can be started by the `ProfileSimulatorOG` module, that simulates the necessary shower data for the FD simulation.

The simulation steps of the example are shown in the `ModuleSequence.xml` file below:

```
<moduleControl>
    <loop numTimes="1">
       <!-- module> EventFileReaderOG       </module -->
       <module> ProfileSimulatorOG          </module>
       <module> EventGeneratorOG            </module>
       <module> FdSimEventCheckerOG         </module>
       <module> ShowerLightSimulatorKG      </module>
       <module> LightAtDiaphragmSimulatorKG </module>
       <module> TelescopeSimulatorKG        </module>
       <module> FdBackgroundSimulatorOG     </module>
       <module> FdElectronicsSimulatorOG    </module>
```

```
      <module> FdTriggerSimulatorOG        </module>
      <module> UserModule                  </module>
   </loop>
</moduleControl>
```

As mentioned above the first module to be used is either `EventFileReaderOG` or `ProfileSimulatorOG`, to read external shower data or to simulate initial shower charge profiles and energy deposit tables, respectively. Note that the `EventFileReaderOG` module can also be used in some SD simulation chains and it is a typical example of a hybrid module of the framework.

The `EventGenerator` generates an impact position and time stamp for the shower core on the ground. Based on the event time and geometry the `FdSimEventChecker` decides which of the telescopes are going to be used for further simulation. The `FdSimEventChecker` uses the Fd uptime manager (by default: `FdAlwaysUpManager`) and the distance and energy dependent telescope trigger propability. the `ShowerLightSimulator` simulates the fluorescence and Cherenkov light along the shower track. The shower light is propagated to the diaphragm of the telescopes by the `LightAtDiaphramSimulatorKG` module. A raytracing module, the `TelescopeSimulatorKG`, simulates the optics of the telescopes and generates photon traces for the PMT's. The photon background is added to the photon traces from the shower light by the `FdBackgroundSimulatorOG` module. The last two physical modules, `FdElectronicsSimulatorOG` and `FdTriggerSimulatorOG`, simulate the FADC traces and the triggers of first and second levels, respectively. Finally, a `UserModule` demonstrates how to generate plots with the simulation results. The plots are stored by the module in the `OfflineOutput.root` file.

4.4 FReconstruction

At present, the FReconstruction example runs a full FD-only reconstruction chain. The module sequence is the following:

```
<moduleControl>
 <loop numTimes="unbounded">
    <module> EventFileReaderOG         </module>
    <module> FdCalibratorOG            </module>
    <module> FdPulseFinderOG           </module>
    <module> PixelSelectorOG           </module>
    <module> FdSDPFinderOG             </module>
    <module> FdAxisFinderOG            </module>
    <module> FdApertureLightOG         </module>
    <module> FdProfileFinderOG         </module>
    <module> FdEnergyFinderOG          </module>

    <!-- Cherenkov subtraction loop -->
    <loop numTimes="10">
        <module> FdCherenkovFinderOG      </module>
```

```
        <module> FdCherenkovSubtracterOG  </module>
        <module> FdProfileFinderOG        </module>
        <module> FdEnergyFinderOG         </module>
      </loop>
      <module> UserModule                </module>
    </loop>
  </moduleControl>
```

When you run with;

```
userAugerOffline -b bootstrap.xml
```

an output file (`OfflineOutput.root`) is created, containing many root histograms: time fit, sdp, light flux at diaphragm, number of particles. They can be viewed with the ROOT browser. As it is distributed, the example is configured to run with fixed calibration constants. To use real calibration constants, comment out the `<configLink>` to `FdCalibrator, FCalibSQLManager, FManagerRegister.xml`. This will tell the system to use the database you specified upon `configure` with the `--with-dbhost=...` option (that is stored in `install-base/share/auger-offline/config/FCalibSQLCon` The default server is `www.hep.physics.neu`. To learn how to get your local database up and running, please refer to `http://www.ifi.unicamp.br/AUGER/projects/dpa/software/distribution/index.html#SE`

Missing: explanation on how to use different atmospheric models,etc.

5 Framework Design and Use

This section provides more details regarding the design of event and detector description, the run control and the configuration machinery. It isn't necessary to digest all these details in order to start using the framework, but you'll probably want to become familiar with most of them over time. For full class documentation, see the doxygen-generated class documentation which comes with the framework.

5.1 Event

In a simulation or reconstruction program, modules read data from and write data to an aggregation of classes representing an event. The event, in fact, involves two distinct layers, an *event interface* and an *event data structure*. Modules communicate with the event interface, which possesses the necessary methods to set and retrieve event information and to check whether desired information exists. The organization of classes in the event interface is meant to be intuitive and is indicated in figure 6. The event data structure, in contrast, is responsible for transferring information between the event interface to some persistent format, such a ROOT file. The idea is illustrated in figure 7. This mechanism is also used to deal with already existing data formats, such as CDAS, FDEventLib and Aires, as well as our own DPA format. Note that the data structure lives "behind the scenes," and, aside from simply specifying the desired format, the user will

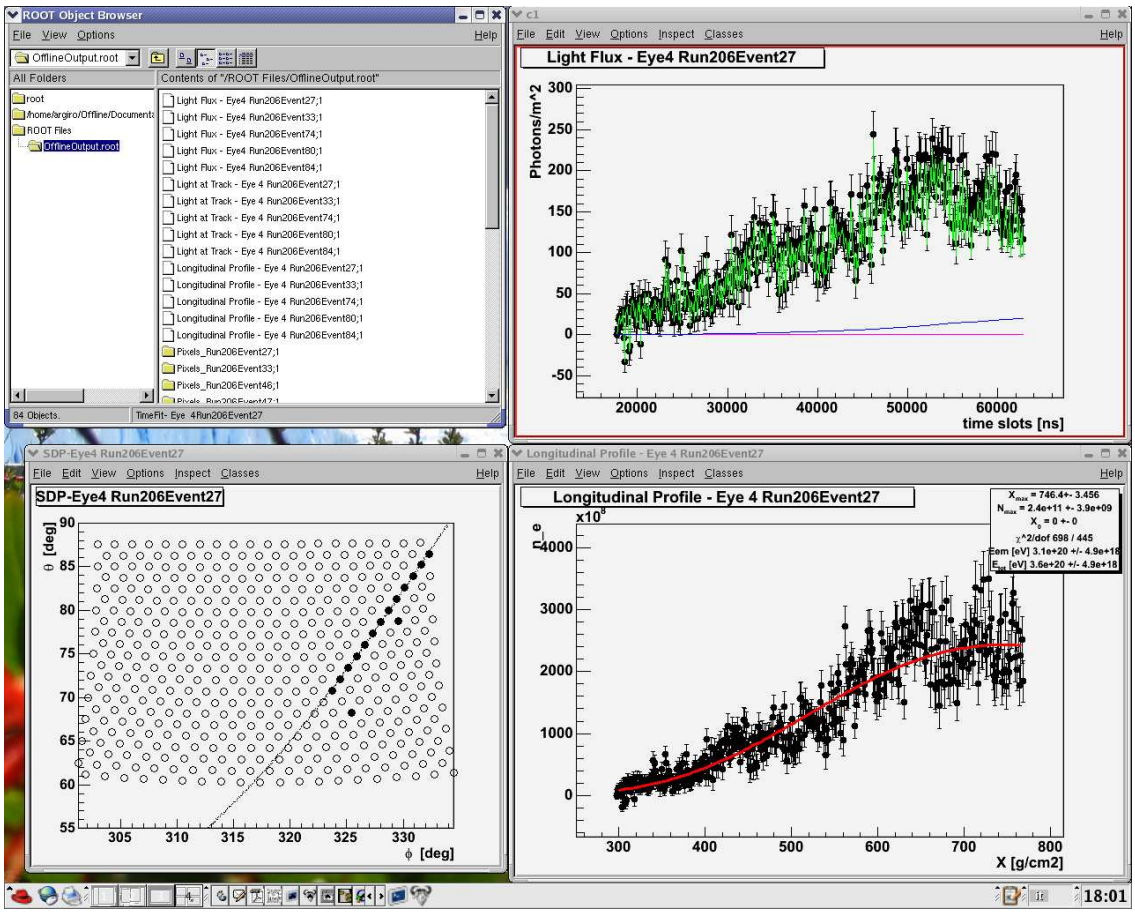


Figure 5: Output of FReconstruction example viewed using a ROOT `TBrowser` object.

not need to deal with it directly. Decoupling of the interface from the data structure allows data formats and, in principle even the persistency mechanism, to be changed without affecting the user code.

5.1.1 Examples of event interface use

The best way to learn your way around the Event interface is to browse the doxygen-generated documentation or look through the examples.

5.2 Detector

Modules can fetch information about the configuration and performance of the detector, including time-dependent information, from the detector description. The detector description provides access to a variety of data sources, including XML files and MySQL databases, through a single interface. The interface comprises a collection of classes which are organized so as to reflect the hierarchy normally associated with the detector. This interface relays the user's request for data to a registry of so-called managers, each of which is designed to deal with a particular data

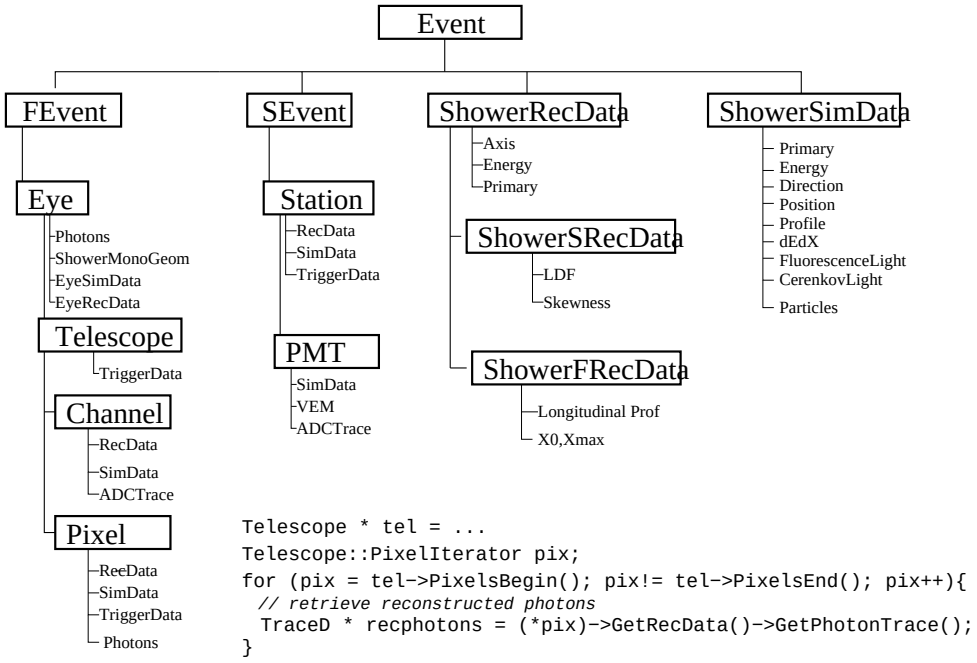


Figure 6: Organization of the event interface.

format. Once a manager locates the requested data, it returns it to the user interface, where it is internally cached so that if another request is made for the same data, it can be more quickly retrieved. The user interface may also perform some simple analysis and/or packaging of the data in to a convenient object before returning it to the interface. Figure 8 illustrates the idea pictorially.

5.2.1 Detector configuration

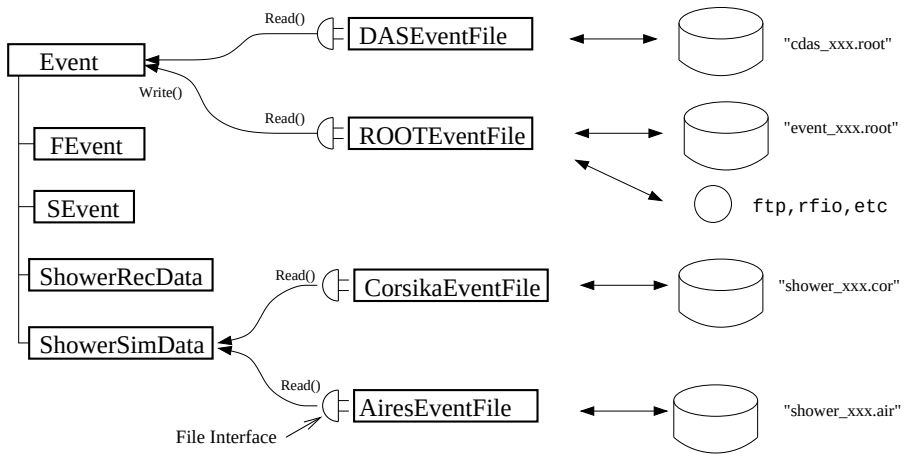
5.2.2 Atmosphere description

5.3 Sequencing Modules and the RunController

The sequencing of modules is controlled by an object known as the `RunController`. There are two ways to give sequencing instructions to the `RunController`. As we have seen in the examples above, a simple module sequence XML file is used to instruct the `RunController` about which modules to use and in what order to use them (see for example section 3.2.2). In addition, individual modules can send instructions via the `fwk::VModule::ResultFlag` return code telling the `RunController` to break a loop, continue a loop, or abort altogether. In addition to controlling module sequences, the `RunController` is responsible for passing the Event reference to the

```
fwk::VModule::ResultFlag Run(evt::Event &)
```

method of each module, as described in section 3.2.1. The module sequence file can also be use to control the state of the Event when it is passed to a module. We illustrate each of these points below.




```
Event event;
DASEventFile infile("cdas_xxx.root");
infile.Read(1,&event);
ROOTEventFile outfile ("ftp://ccali.in2p3.fr/event_xxx.root", "w");
event.Write(outfile);
```

Figure 7: Event interface (left side) and event data structure (right side). Users only interact directly with the event interface, which provides access to all available information about the event. The event data structure employs so-called File objects which are able to move data back and forth between various on-disk formats (eg. CDAS, FDAS, Offline native, Aires, Corsika) and the event interface. This insulates the user from the various on-disk formats; if for some reason these formats change from one version to the next, the physics code will not have to be adapted to the change.

First, to illustrate basic looping instructions as well as the `fwk::VModule::ResultFlag` return code, assume we run the module sequence shown in Figure 9. Let's consider the effect of ModuleC returning various values of `fwk::VModule::ResultFlag` from its `Run(Event &)` method. The `ResultFlag` is simply an enumeration which can take four possible values: `eSuccess, eFailure, eContinueLoop, eBreakLoop`. For the case `eSuccess`, nothing interesting happens; it is assumed that all went well for ModuleC, and the `RunController` carries on to ModuleD. In contrast, if ModuleC returns `eFailure`, it is assumed that a catastrophic error occurred, and execution is immediately terminated by the `RunController`. If ModuleC returns `eBreakLoop`, then the control is returned to the loop level outside of the current level, or, if there is no such outer loop, execution is terminated. In the example above, an `eBreakLoop` returned by ModuleC would cause the RunController to jump out of the inner loop and run ModuleA. (Notice also that in the example above, the outermost loop happens an "unbounded" number of times; this implies a never-ending loop unless ModuleA returns `eBreakLoop` or unless any of the modules returns `eFailure`). Finally, if ModuleC returns `eContinue`, all subsequent modules at the current loop level are skipped and control returns to the beginning of that loop. In the example above, this would imply ModuleD would be skipped and ModuleB would be the next one executed. This sort of behavior is useful when, for example, you are looping on events and applying selection criteria

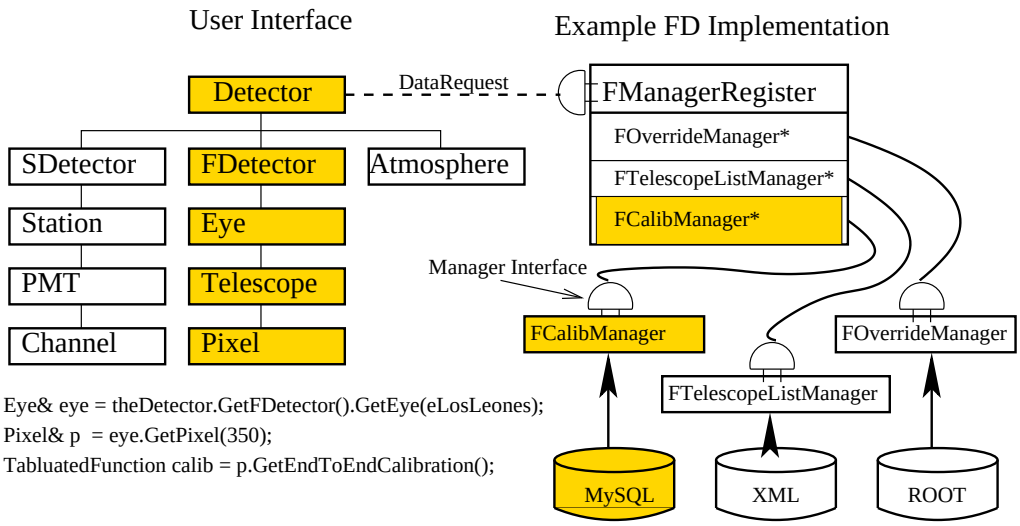


Figure 8: Machinery of the detector description. The user interface is indicated on the left and comprises a number of classes representing the detector components. Requests for data are relayed to a registry of Managers (FD managers are shown in the figure). Each manager knows how to deal with a particular data source, such as ASCII or XML files or MySQL databases. When one of the managers locates the requested data, it sends it back to the user interface, where it is cached for later use. In the example depicted here, the user requests the end-to-end calibration for a particular pixel. The request for these data are relayed to the FManagerRegister, which contains several managers. One of these managers is able to fetch the calibration data, and relay them back to the Pixel object of the user interface, which in turn packages up the data in a TabulatedFunction describing the calibration constant as a function of wavelength.

as a condition for continued processing of the event.

Now we describe how the instructions in the module sequence file can control the state of the event passed via the `fwk::VModule::ResultFlag Run(evt::Event &)` method. There are two different sorts of behavior that are typically desired, depending on the application.

First consider an FD reconstruction sequence in which Cerenkov light subtraction is carried out through an iterative procedure. A module sequence for such an application looks something like:

```
<loop numTimes="unbounded">

  <module> EventFileReader </module>
  <module> GeometryFinder  </module>
  <module> ProfileFinder   </module>
  <module> EnergyFinder    </module>  <!-- 1 -->

  <loop numTimes="10">  <!-- iterative Cerenkov subtraction -->
    <module> CerenkovFinder     </module>
    <module> CerenkovSubtracter </module>
    <module> ProfileFinder      </module>
    <module> EnergyFinder       </module>  <!-- 2 -->
```

```
<loop numTimes="unbounded">
  <module> ModuleA </module>
  <loop numTimes="10">
    <module> ModuleB </module>
    <module> ModuleC </module> <!-- returns a ResultFlag -->
    <module> ModuleD </module>
  </loop>
</loop>
```

Figure 9: A module sequence used to illustrate effect of `fwk::VModule::ResultFlag` (see text).

```
  </loop>

  <module> Analysis </module>
</loop>
```

In this case, the desired behavior is as follows. The first time the CerenokvFinder module is run, the Event should be in the state that the EnergyFinder at `<!-- 1 -->` left it. The subsequent 9 times the CerenkovFinder is run, the Event should be in the state that the EnergyFinder at `<!-- 2 -->`. That is, passing through the `<loop>` tag should not affect the state of the event.

Now consider a SD simulation example. A module sequence to generate 10 fully simulated showers (including the detector response) from a single Aires or Corsika shower would look something like:

```
<loop numTimes="unbounded">
  <module> EventFileReader </module>     <!-- reads in an Aires or Corskia shower -->
  <loop numTimes="10" save="yes">
    <module> EventGenerator    </module> <!-- throws sim shower somwhere on array -->
    <module> SimulatorModules  </module> <!-- actually involves several modules -->
    <module> EventFileExporter </module>
  </loop>
</loop>
```

In this sequence the desired behavior is different from the behavior in the previous example. Here, there is an outer loop which reads in several Aires or Corsika showers from one or more files. For each shower, there is an inner loop in which we want a fully simulated shower to be constructed from the Aires or Corsika shower. Thus, the EventGenerator module expects to always see an Event which is in the state the EventFileReader left it (not in the state that the the EventFileExporter left it!). To produce this sort of behavior, we add the `save="yes"` attribute to the inner `<loop>` tag. This save attribute tells the RunController to store the event just before entering the `<loop>`, then restore the event in that saved condition just prior to each execution of the first module inside the `<loop>`. This event saving option works like a stack, so that it can be applied in loops nested with arbitrary depth.

5.4 Configuration files and CentralConfig

In order to provide the flexibility to carry out many different sorts of applications, the $\overline{\text{Off}}\underline{\text{line}}$ framework was designed to be highly configurable. One consequence of this is that there can be many configuration files to keep track of! A portion of the framework known as the `CentralConfig` provides services to specify which configuration files should be used, import default configurations, and log the configurations used for a run. The `CentralConfig` object is essentially just a dictionary that associates a file on a local disk or a URL on the internet to a logical name. Framework code or user code can then request configuration information using the logical name, and the `CentralConfig` will look up the appropriate physical file.

Configuration files are typically required for such tasks as:

- setting up the detector description,
- sequencing modules (see section 3.2.2), and
- configuring individual modules. Module configuration files could include any data which does not naturally belong to the detector description, such as cuts or parameters used to tune a particular algorithm.

5.4.1 Using the CentralConfig

The `CentralConfig` fills itself using a so-called bootstrap XML file whose name is passed on the command line of the main program. Figure 10 shows an example of a bootstrap XML file. Once constructed, the `CentralConfig` object evaluates everything between the `<centralConfig>` tags of the bootstrap file. Currently, it only responds to information it finds in the attributes of `<configLink>` tags. These attributes define a link `id`, the `type` of file, and and `xlink:href`. These three items are discussed below.

The `configLink id` attribute is simply a logical name used from within C++ code to refer to a link, where a link in this context is simply a file path and name or a URL. For example, to request the file name and path for `Module1Config.xml` (see Figure 10), one could query the `CentralConfig` as follows:

```
CentralConfig* theCC = CentralConfig::GetInstance();
string module1File = theCC->GetLinkName("Module1Config");
assert (module1File == "./Module1Config.xml");
```

The `type` attribute contains information about the file format, and is used by the `CentralConfig` to take some action based on what format it is dealing with. For example, if `type = "XML"`, the `CentralConfig` will automatically create a `Reader` [2] for the requested XML file. The user can then jump directly into the XML document using the `GetTopBranch` method of `CentralConfig`:

```
CentralConfig* theCC = CentralConfig::GetInstance();
Branch topOfModule1Config = cc->GetTopBranch("Module1Config");
```

[2] For documentation on the `Reader` class, see reference [6].

```
<bootstrap xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation='/someplace/bootstrap.xsd'
                xmlns:xlink="http://www.auger.org/schema/types">

  <centralConfig>

    <!-- The file containing the sequencing file used by RunManager -->
    <configLink
        id         = "ModuleSequence"
        type       = "XML"
        xlink:href = "./ModuleSequence.xml" />

    <!-- A file containing parameters used by one of the modules -->
    <configLink
        id         = "Module1Config"
        type       = "XML"
        xlink:href = "./Module1Config.xml" />

    <!-- A file containing SD detector geometry and materials for simulations -->
    <configLink
        id = "SModelsXMLManager"
        type = "XML"
        xlink:href = "http://www.hep.physics.neu.edu/auger/xml/SModelConfig.xml" />

  </centralConfig>
</bootstrap>
```

Figure 10: Example of the bootstrap.xml file, which contains links to various configuration files.

(Branches are described in more detail in section 5.5.) On the other hand, if `type = "ROOT"`, for example, no `Reader` is created, as the `Reader` class cannot read ROOT files. In this case, one could only request the file name and path using the `GetLinkName` method; the `GetTopBranch` method would return a `NULL Branch`.

Finally, we comment on the `xlink:href` attribute. As discussed above, this attribute contains the actual link to the file, where in our case this link is just a file name and path or a URL. Using a URL is sometimes convenient in cases where you are collaborating on a task and wish to use a common configuration file. One person can post and maintain the configuration on a web site, and everyone else can simply set their configLink to the appropriate URL.

5.4.2 Configuration logging

The CentralConfig also contains machinery to assist in logging the complete configuration used during a run. This feature is invoked using the `-l` option on the command line. For example,

```
userAugerOffline -b bootstrap.xml -l logfile.xml
```

or simply,

```
userAugerOffline -b bootstrap.xml -l
```

will cause the CentralConfig to keep track of all configuration files requested during the run and, at the end, concatenate them into one big XML log file. You can specify a log file name, as in the first example above, or let the CentralConfig assign one, as in the second example. This log file can be opened in any text editor, or it can be viewed in an XML-enabled browser (eg. recent versions of Mozilla work well). If you look through a log file, you'll discover that it begins with something that looks like a bootstrap file. Rather than pointing to external files, this bootstrap file points to different locations within the same log file in which the various configurations are located. As a result, you can use a log file as if it were a bootstrap file, and so reconstitute a run with exactly the same configuration that was originally used. Thus the command,

```
userAugerOffline -b logfile.xml
```

will run the program with precisely the configuration used at the time the `logfile.xml` file was generated.

5.4.3 Default configurations

The bootstrap file for a typical simulation or reconstruction application can get rather long, particularly if there are many modules and/or data sources. To make life a bit easier, default configurations are available. Currently there is one SD default configuration and one FD default configuration; you can use them together if you're doing hybrid work.

The default configurations can be included in your bootstrap file, as illustrated in figure 11. Including a default configuration involves declaring a so-called `ENTITY`, as shown at the top of the figure. In XML-speak, an `ENTITY` simply defines an alias that gets replaced by some other

text at the time the file is parsed. In this case, `defaultSDConfig` gets replaced by a bit of XML (defining the default configuration) which is read from a file. The magic word `SYSTEM` tells the parser that the XML we are interested in resides in a separate file, rather than in the current document. The name and location of this file is given following the `SYSTEM` keyword; in this case, the file is `/someplace/defaultSDConfig.xml`. When the bootstrap file is parsed, the parser replaces `&defaultSDConfig;` with the XML read in from `defaultSDConfig.xml`.

```
<!DOCTYPE bootstrap [
  <!-- alias for default SD config files -->
  <!ENTITY defaultSDConfig SYSTEM '/someplace/defaultSDConfig.xml'>
]>

<bootstrap xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation='/someplace/bootstrap.xsd'
               xmlns:xlink="http://www.auger.org/schema/types">

  <centralConfig>

  <!-- get default configuration for SD -->
  &defaultSDConfig;

  <configLink
      id         = "ModuleSequence"
      type       = "XML"
      xlink:href = "./ModuleSequence.xml" />

  <configLink
      id         = "EventFileReader"
      type       = "XML"
      xlink:href = "./EventFileReader.xml" />

  <configLink
      id         = "EventFileReader"
      type       = "XML"
      xlink:href = "./EventFileReader.xml" />

  <!-- override default ParticleInjector configuration -->
  <configLink
      id         = "ParticleInjector"
      type       = "XML"
      xlink:href = "./myParticleInjectorConfig.xml" />

  </centralConfig>
</bootstrap>
```

Figure 11: Example of including a default SD configuration in a bootstrap file.

Notice in Figure 11 that, despite including a default configuration, we still define `configLink`'s for the ModuleSequence and EventFileReader logical names. This is because the `defaultSDConfig.xml` file does not set default configurations for these, as there are no sensible default module sequences

of event file reader configurations.

You can easily override a `configLink` set by a default configuration file by simply redefining that configLink in your bootstrap file anywhere below the inclusion of the default configuration. In Figure 11, for example, the default configuration for the ParticleInjector module is replaced with a different configuration file.

5.5 Reading an XML file from your module

The various framework configuration files use XML format. The rationale for using XML, rather than some homemade ASCII format, is that it is a universally adopted standard which is designed to be both a human readable and machine parsable way of organizing data, and, as such, there are several well written XML parsers which are freely available. As part of the Offline code, we provide a utility called `Reader` which is based on the Xerces parser [7], and provides ease-of-use and the cost of a small reduction in functionality compared to Xerces. See [6] for details on the Reader code as well as a brief XML tutorial and further information in the advantages of using XML.

You may want to have a configuration file for your module in order to store values you wish to vary, such as cuts or other parameters. In such a case you can, of course, use the Reader to parse it. Here we give an example of how to do this. This example is probably more involved than a typical module configuration, but it serves to illustrate everything you are ever likely to need to know concerning navigation through a hierarchy of data.

5.5.1 Accessing your configuration file

You can tell the CentralConfig where your configuration file lives by adding a `<configLink>` tag in the `bootstrap.xml` file. The information in the file can then be looked up from within your code. See section 5.4 for an explanation of how to do this.

5.5.2 Navigating Data Organized in Trees

For purposes of illustrating how to navigate through a tree of data, we consider the following example XML file.

```
<?xml version="1.0" ?>

<document>

<!-- Example detector simulation parameters - simplified
for illustrative purposes -->

<detectorSimParameters>

  <tank id="1">
    <radius unit="m"> 1.8 </radius>
    <height unit="m"> 1.2 </height>
    <PMT id="1">
      <position unit="cm"> 0.0 120.0 60.0 </position>
      <maxQe> 0.30 </maxQe>
    </PMT>
```

```
    <PMT id="2">
      <position unit="cm"> 103.92 -60.0 60.0 </position>
      <maxQe> 0.30 </maxQe>
    </PMT>
    <PMT id="3">
      <position unit="cm"> -103.92 -60.0 60.0 </position>
      <maxQe> 0.30 </maxQe>
    </PMT>
  </tank>

  <tank id="2">
    <PMT id="3" >
      <position unit="cm"> -103.92 -60.0 60.0 </position>
      <maxQe> 0.27 </maxQe>
      <photonEnergyBin unit="eV">
        2.14  2.16  2.19  2.23  2.27  2.32
      </photonEnergyBin>
    </PMT>

    <tyvekProperties>
      <reflectivity>  0.9164 </reflectivity>
      <specularLobe>  0.2 </specularLobe>
      <specularSpike> 0.0 </specularSpike>
    </tyvekProperties>
  </tank>

</detectorSimParameters>

</document>
```

Note the information is organized in a heirarchical "tree" structure. In Reader parlance, each element in the XML document constitutes a *Branch*. For example, in the document above `tank` is a Branch, which has child Branches `PMT` and `tyvekProperties`. The Branch `PMT`, in turn, has child Branches `position`, `maxQe` and `photonEnergyBin`. Note that a Branch can contain either more branches, data, or both.

The begin tag for a Branch may also have (optional) "attributes." In the examples above, one attribute that appears is called `id` and another is called `unit`. The `unit` is meant to set the unit for data contained in a branch, and is treated in a special way, as discussed below.

Finding the Top of the Document The first step in navigating a tree is to request the top branch of the XML document from the Reader. If we use the `CentralConfig` to look up the XML file (section 5.4.1, we can retrieve the top Branch of the document as follows:

```
CentralConfig* theCC = CentralConfig::GetInstance();
Branch topB = cc->GetTopBranch("someConfigFile");
```

In our example, this will return the `document` branch. You can verify this using the `GetBranchName` method:

```
cout << topB.GetBranchName() << endl;
```

Getting the Desired Branch Starting from the top branch, you can find any of the children. A child branch can be retrieved by name as in this example:

```
Branch simB = topB.GetChild("detectorSimParameters");
```

If you want to look up a Branch which has one or more attributes associated with it, you can load up the desired attributes into `map` and pass it as an argument of the `GetChild` method:

```
map<string, string> atts;
atts["id"] = "1";
Branch heightB = simB.GetChild("tank",atts);
```

Casting Data to the Desired Type One of the tasks the Reader performs for you is casting the data in a branch to the requested type. Currently the following types are supported [3]:

```
bool, int, float, double, char*, string
vector<bool>, vector<int>, vector<float>, vector<double>, vector<string>
list<bool>, list<int>, list<float>, list<double>, list<string>
```

Casting to the desired type is handled by the `GetData` method of `Branch`. This method is overloaded with argument lists corresponding to the different supported types. For example, to return a double corresponding to the `height` data for the `tank` with `id="1"`, one could write:

```
double height;
heightB.GetData(height);
```

where the `heightB` branch is set in the examples above. Retrieving data into an STL container like `list` or `vector` is also straightforward, as in the following example:

```
map<string,string> tankAtts;
tankAtts["id",2];
map<string,string> pmtAtts;
pmtAtts["id",3];

list<float> energy;
simB.GetChild("tank",tankAtts).GetChild("PMT",pmtAtts).
     GetChild("photonEnergyBin").GetData(energy);
```

The `simB` branch, which points to `detectorSimParameters`, is set in the examples above.

The Reader does not generally protect against user foolishness. If you try to cast some text to an int, for example, the Reader will do its best and return an int. There are other mechanisms to protect against such mistakes, however, as discussed in section 5.5.3.

Dealing with Units As mentioned previously, the Reader provides a mechanism to handle units associated with data in an XML file. Units may be declared via a special "unit" attribute, which appears inside the begin tag. In the XML file above, there are several examples of this.

When data from the XML file is cast via the `GetData` method, any unit attribute which may be present in the tag is evaluated and the data between the begin and end tags is multiplied by whatever factor is necessary to convert it into Auger base units. In this way you can write data in whatever units are the most convenient, and the Reader code will convert them back into "official" units. The Reader supports units expressions, so for example you could write something like:

```
<g unit="m/s^2"> 9.8 </g>
<aperture unit="sr*km2"> 3000 </aperture>
```

The Auger base units are defined inside the `AugerUnits.h` include file, which is a compilation of factors to convert dimensional quantities into official Auger units. For example, some of the `AugerUnits.h` conversion factors for length read:

```
static const double meter  = 1.0;
static const double m  = meter;
static const double millimeter  = 1.e-3*meter;
static const double mm  = millimeter;
```

[3] If you need support for additional types, please contact the authors.

As an example of how one uses these conversion factors, suppose you want to read in the following branch of the example XML file:

```
<radius unit="m"> 1.8 </radius>
```

This could be done with the following lines:

```
double radius;
simB.GetBranch("tank","1").GetBranch("radius").GetData(radius);
```

Then, to print the `radius` variable in millimeters, one could use the appropriate conversion factor from AugerUnits.h, as in the following code:

```
cout << "radius = " << radius/mm << endl;
```

5.5.3 Validating configuration files

It's obviously a good idea to check data read in from XML files for garbage and typos before using it. Traditionally, this has been done by writing a lot of code to verify that each required piece of data is found in the file and that it is set to some reasonable value. For instance, if you browse through various modules, you may encounter snippets of code like this:

```
Branch dataB = topB.GetChild("data");
if (!dataB) {
  ERROR("Could not find requested data");
  return eFailure;
}
double fData;
dataB.GetData(fData);
```

While this bit of code is better than nothing, it is not particularly comprehensive, as it only checks for existence of a required Branch, but doesn't do anything to verify that the data contained therein are sensible. Writing more detailed validation tends to be tedious and error prone. Most people cannot be bothered.

A better approach is to exploit one of the standard methods for XML document validation. Validation in this case means employing an auxiliary ASCII file (rather than C++ code) which sets down rules that a particular XML document must obey. When the XML document is parsed, it is checked against these rules. There are several validation standards, and we have adopted one known as Schema for most of the framework's internal configuration files. Schema files are just XML files with specially defined tags that can be used to place requirements on the structure and contents of the XML file. While Schema is very comprehensive and can get involved for complex applications, most module configuration files are simple enough that it is generally quite easy to prepare Schema files to validate them. Although it isn't necessary to use Schema for your physics modules, it generally leads to more robust software, and is therefore worth considering.

As an example, Figure 12 shows simple XML file and its corresponding Schema validation file. Notice first that each element in the XML file is declared in the Schema file by a line beginning with `<xs:element ....` The `xs:` prefix is just a declaration of the Schema standard namespace, which is analogous to the `std` namespace in C++. The first element to be declared in the Schema is `aConfigFile`, the outermost element in the XML file. Next, the `<xs:complexType>` tag indicates that contents of the `aConfigFile` element are "complex," meaning that these contents can comprise, among other things, a number of sub-elements. In this example the sub-elements of `aConfigFile` in the XML file are `waterRIndex`, `dEdXMuon`, and `PhotonIntLength`. In the Schema, each of these three sub-elements is declared to be of a particular type; `WaterRIndex` is declared as a double, `dEdXMuon` is a double with a unit attached to it, and `PhotonIntLength` is a list of doubles with units attached to each. Notice that `doubleWithUnit` and `listOfDoublesWithUnit`

```
--------------------- XML data file ----------------------------------

<aConfigFile  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:noNamespaceSchemaLocation='someplace/aConfigFile.xsd'>

  <WaterRIndex>              1.33  </WaterRIndex>
  <dEdXMuon unit="GeV">      0.2   </dEdXMuon>
  <PhotonIntLength unit="m"> 0.06 0.07 0.08 0.1 0.21 </PhotonIntLength>

</aConfigFile>

---------------------- Schema validation file -------------------------

<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'
           xmlns:auger="http://www.auger.org/schema/types">

  <xs:import namespace="http://www.auger.org/schema/types"
             schemaLocation="/someplace/AugerSchemaTypes.xsd"/>

  <xs:element name="aConfigFile">
    <xs:complexType>
      <xs:all>
        <xs:element name="WaterRIndex" type="xs:double"
                    minOccurs="1" maxOccurs="1"/>
        <xs:element name="dEdXMuon" type="auger:doubleWithUnit"/>
        <xs:element name="PhotonIntLength"
                    type="auger:listOfDoublesWithUnits"/>
      </xs:all>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Figure 12: Example XML file (top) and Schema file used to validate it (bottom).

types live in a namespace called `auger:` rather than `xs:`. This is because these types are not standard, but are defined in a file distributed with the framework, as they prove useful for some of our applications. The line in the schema file which begins `<xs:import namespace=....>` is an instruction to import these special Auger data types from the file where they are defined. The `minOccurs` and `maxOccurs` attributes appearing in the declaration of the `WaterRIndex` element specify that this element must appear exactly once. Finally the `<xs:all>` tags surrounding the declarations of the three elements sets the requirement that all three of the declared elements must be found in the XML file.

When the XML file of Figure 12 is parsed, it is checked against the Schema file, whose location is specified by the `xsi:noNamespaceLocation='...` attribute of the `<aConfigFile...` element. Any violations of the rules set forth in the Schema result in an error message specifying the nature of the offense and line and column where it occurs. For example, the following mistakes would be caught during validation:

```
<WaterRIndex> 1.33xs  </WaterRIndex>           <!-- stray characters -->
<WaterRIndex unit="EeV> 1.33  </WaterRIndex>   <!-- should not have a unit -->
<dEdXMuon unit="GeV">  0.2 0.3   </dEdXMuon>   <!-- only one double allowed -->
<dEdXMuon> 0.2 </dEdXMuon>                     <!-- forgot the unit -->
<PhotonIntLength unit="m">  </PhotonIntLength> <!-- forgot the data -->
```

Furthermore, if any of the three declared elements is missing from the XML file, the `<all>` condition would be violated, and a corresponding error reported. Programming this level of error checking on your own would require significantly more typing (and debugging!) than preparing a Schema file.

If you retrieve the top Branch of a configuration file using the `CentralConfig`, in the manner described in section 5.4.1, then Schema validation will be performed if (and only if) you specify a schema file using the `xsi:noNamespaceSchemaLocation` attribute of the top element of your XML document. If you leave that attribute out, or if the file it points to is nonexistent, then no validation will be performed. If you use the `Reader` utility directly to open and parse and XML file, they you can explicitly switch on and off validation. See the doxygen documentation for the `Reader` class for details.

More detailed explanations and tutorials related XML and Schema abound on the WWW.

5.6 Databases

5.7 Geometry

5.8 Error logging

5.9 Time stamps and time intervals

References

[1] S. Argiro, *et al.*, "Proposed design for Auger DPA Production Software" GAP-2002-021.

[2] S. Argiro and A. de Capoa, "A prototype event structure and implementation for Auger offline software" GAP-2002-026.

[3] T. Paul, "Prototype framework for Auger offline software" GAP-2002-066.

[4] T. Paul and J. Macleod, "Prototype detector description and database handling for Auger offline software" GAP-2002-052.

[5] Markus Roth for the Pierre Auger Collaboration, "The Lateral Distribution Function of Shower Signals in the Surface Detector of the Pierre Auger Observatory", 2003, Proc. 28th Int. Cosmic Ray Conf. (Tsukuba), HE1.1, 333.

[6] T. Paul, "Tools for parsing data" GAP-2001-044.

[7] `http://www.apache.org`

[8] T. McCauley and T. Paul, "GEANT4 simulation of the surface detectors" GAP-2000-055.

A Appendix: The Offline Build System

B Appendix: Installation and configuration of the GEANT4-based SD simulation

In previous incarnations the GEANT4-based SD simulation [8] has existed as a stand-alone, single-tank simulation and as part of a full-array simulation in various prototype frameworks. The latest, supported version now resides in the Offline installation as an optional module in `Offline/Modules/G4TankSimulatorOG` .

In the default configuration of the Offline software the GEANT4-based SD simulation is not compiled and installed with the rest of the package. If one wishes to install it, one must of course make sure that first GEANT4 is installed. The source code (and if one wishes, precompiled binaries) can be downloaded from the GEANT4 webpage:

`http://geant4.web.cern.ch/geant4` .

For a standard installation, the following environmental variables must be set [4]:

- **G4SYSTEM** (`Linux-g++`)
- **G4INSTALL** (location where Geant4 is installed)
- **CLHEP_BASE_DIR** (location where CLHEP is installed)

Typing "make" in `$(G4INSTALL)/source` will begin the build. When the build is (finally) finished, typing "make includes" will copy all the header files to `$(G4INSTALL)/include`. Typing "make install" is not required as the libraries are installed in `$(G4LIB)` [5] when the build is completed. By default static, granular libraries are built. If one wishes to have global libraries, then one must type "make global" instead of "make" to begin the build. Dynamic libraries will be built if `$(G4LIB_BUILD_SHARED)` is set. Additional options such as verbosity, optimization, and debugging can be set as well; consult the GEANT4 webpage for more information.

The `G4TankSimulator` module can be included in the Offline software build by specifying the location of the GEANT4 installation with the configure script on the command line:

```
./configure --with-geant4=$G4INSTALL
```

The installation of the Offline software can then proceed as detailed in this document. The module `G4TankSimulator` module can be selected as any other module included in the Offline release.

[4] GEANT4 depends on CLHEP, and it is important to have the right combination of release versions. The latest version (as of this writing) of GEANT4 (6.1) requires that you have at least CLHEP1.8.0.0.

[5] By default, `$(G4LIB)` is set to `$(G4INSTALL)/lib/$(G4SYSTEM)`